LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
I l 6r
no. 308-315
cop. 2

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

**Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.**

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

APR 26 1974

APR 9 Rec'd

L161—O-1096

Digitized by the Internet Archive
in 2013

http://archive.org/details/tranquillanguage315abel

Math
I l6r
no. 315
cop. 2

Report No. 315

# TRANQUIL: A LANGUAGE FOR AN ARRAY PROCESSING COMPUTER

by

Norma E. Abel, Paul P. Budnik, David J. Kuck
Yoichi Muraoka, Robert S. Northcote, Robert B. Wilhelmson

April 1, 1969

ILLIAC IV Document No. 215

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 315

# TRANQUIL: A LANGUAGE FOR AN ARRAY PROCESSING COMPUTER

by

Norma E. Abel, Paul P. Budnik
David J. Kuck, Yoichi Muraoka
Robert S. Northcote, Robert B. Wilhelmson

April 1, 1969

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

* This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. US AF 30(602)4144 and in part by the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, April 1, 1969.

## ABSTRACT

TRANQUIL, a programming language for the description of algorithms for an array processing computer, such as ILLIAC IV, and the implementation of its compiler is discussed. The language, the structure of which is based on Algol, allows the use of index sets, which may be used to specify which processing elements of the array are to be activated for a given operation. There are also new types of control statements called SIM and SEQ which, used in conjunction with index sets, facilitate the explicit specification of parallelism in the algorithm. Array operators are also allowed in the language. Some of the special compilation techniques needed to generate code for the array computer ILLIAC IV are also discussed.

# 1. INTRODUCTION

TRANQUIL is the algorithmic language which will be used to write programs for ILLIAC IV, a parallel computer which has been described by Barnes et al[1]. ILLIAC IV is designed to be an array of 256 coupled processing elements (PE's) arranged in four quadrants in each of which the 64 PE's are driven by instructions emanating from a single control unit (CU). Each of the 256 PE's is to have 2048 words of 64 bit semiconductor memory with a 250 nanosecond cycle time and an instruction set which includes floating point arithmetic on both 64 bit and 32 bit operands with options for rounding and normalization, 8 bit byte operations, and a wide range of tests due to the use of addressable registers and a full set of comparisons. The PE's differ from conventional digital computers in two main ways. Firstly, each is capable of communicating data to its four neighboring PE's in the array by means of routing instructions. Secondly, each PE is able to set its own mode registers, thus effectively enabling or disabling itself for the transmission of data or the execution of instructions from its CU.

Figure 1 shows 64 PE's, each having three arithmetic registers (A, B, and C) and one protected addressable register (S). The registers, words, and paths in Figure 1 are all 64 bits wide, except the PE index registers (XR), mode registers, and as noted. The mode register may be regarded as one bit which may be used to block the participation of its PE in any action. The routing

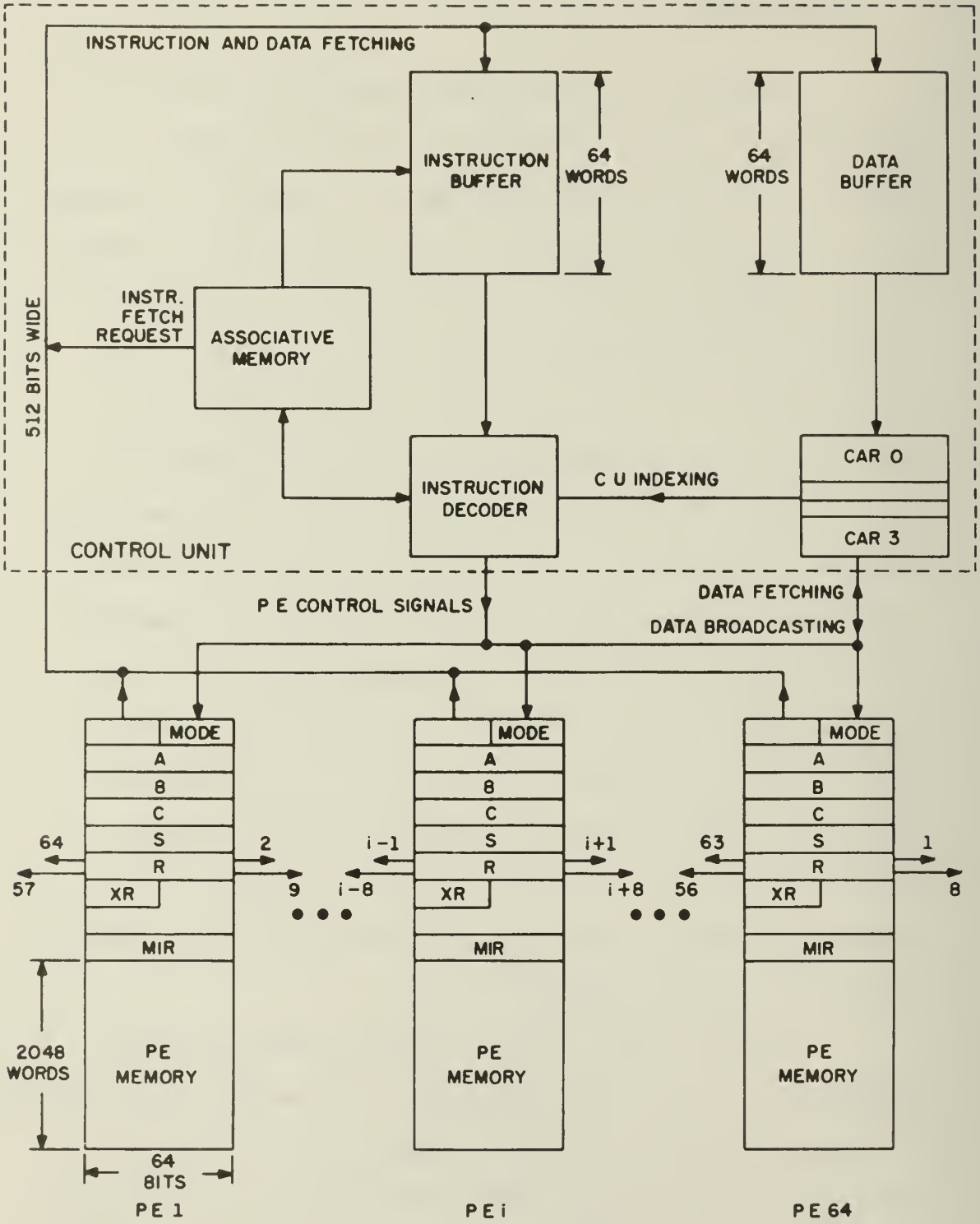


Figure 1. ILLIAC IV quadrant configuration

registers are shown connected to neighbors at distances $\pm 1$ and $\pm 8$; similar end around connections are provided at 1, 64, etc. Programs and data are stored in PE memory. Instructions are fetched by the CU in blocks of 8 words as required and are stored in a 64 word CU instruction buffer.

Figure 1 also shows the essential registers and paths in the CU and their relations to the PE's. Instructions are decoded and control signals are sent to the PE array from the control unit. Some instructions are executed directly in the CU; e.g., the loading of CU accumulator registers (CAR's) with program literals. Operand addresses may be indexed once in the CU and again separately in each PE. It is possible to load the local data buffer (64 words of 64 bits each) and CAR's from PE memory. Local data buffer registers and CAR's may be loaded from each other. A broadcast instruction allows the contents of a CAR to be transmitted simultaneously to all PE's. It is often convenient to manipulate all PE mode bits or a number from one PE in a CAR. For this purpose, the broadcast path is bidirectional.

The four control units may be operated independently, as pairs, or all together. In the united configuration, all 256 PE's are effectively driven by one CU and routing proceeds across PE boundaries. This allows some flexibility in fitting problems to the array.

If ILLIAC IV, or any other parallel array computer, is to be used effectively, it is essential that all possible parallelism be detected in those algorithms which are to be executed by that computer.

This is difficult, if not impossible, if the algorithms are specified in languages such as FORTRAN and ALGOL which essentially express all computational processes in terms of serial logic, as required for conventional computers. Since it is also more convenient for the user to express array type computation processes in terms of arrays and parallel operations, rather than having to reduce the inherent parallelism to serial computational form, the specification of a new language for array processor computation is clearly necessary.

The TRANQUIL language has been designed to achieve both simpler specifications of, and explicit representation of the parallelism in, many algorithms, thus simplifying the programmer's task and maximizing the efficiency of computation on a computer such as ILLIAC IV. An overview of the software and application programming effort for the ILLIAC IV system has been given by Kuck[2].

## 2. THE TRANQUIL LANGUAGE

An important consideration in designing a language such as TRANQUIL is that the expression of parallelism in the language should be problem oriented rather than machine oriented. This does not, and should not, preclude programmer specification of data structure mapping at run time, but once the storage allocation has been made the programmer should have to think only in terms of the data structures themselves. Secondly, the means of specifying the parallelism should be such that all potential parallelism can be specified.

The structure of TRANQUIL is based on that of ALGOL; in fact, many ALGOL constructs are used with the addition of further

data declarations, standard array operators, and revised loop specifications, including the addition of the set concept. Some of the ideas embodied in TRANQUIL follow similar constructs in other languages, e.g., the index sets in MADCAP[3] and the data structures and operators in APL[4]. The syntax of the current version of the TRANQUIL language is specified in Appendix B.

## 2.1 Data Structures

The data structures which are recognized in TRANQUIL are simple variables, arrays and sets. All data structures, and quantities such as labels, switches and procedures, must be declared in some block head as in ALGOL. The data type attributes are INTEGER, REAL, COMPLEX and BOOLEAN. Certain precision attributes also may be specified.

A mapping function specification must be associated with every declaration of an array. The judicious choice of mapping functions is crucial to the efficient use of ILLIAC IV. Arrays must be mapped so as to optimize I/O transactions, minimize unfilled wasted areas of memory, and keep most of the PE's busy most of the time. In many array operations it is necessary to operate either on a whole row or a whole column of an array. All the PE's would be kept busy in the former case (one operand in each PE) but in the latter case all operands would normally be in only 1 PE. However, by specifying the skewed mapping function which rotates the i + 1st row across i PE's, columns as well as rows of the array can be accessed simultaneously. The more commonly used mapping functions such as STRAIGHT, SKEWED, SKEWED PACKED, and CHECKER are

included in TRANQUIL. Array bounds may be specified dynamically, as in ALGOL, but all other attributes are nondynamic, for example:

```
REAL SKEWED ARRAY A[1:M, 1:N]
```

The user who wishes to specify his own mapping function may make use of a PE memory assignment statement. For example:

```
PEMEMORY PB [1:10, 1:64 ];
PEM FOR (I, J) SIM ([1, 2, ..., 10] × [1, 2, ..., 64 ]) DO
     PB [I, J] ← B [I, MOD ( 64, I + J - 1)];
REAL ARRAY  (PB) B[1:10, 1:64 ];
```

where SIM is discussed in section 2.3.2, establishes virtual space of size $10 \times 64$ in PE memory, and then stores a $10 \times 64$ array B there in skewed form. Thus, instead of making up the aforementioned subarrays out of an array declaration, space reserved in PE memory may be used. In the program, the programmer refers to an element of memory space via the assigned array name B and its subscripts, as usual. It should be noted that storage mapping functions can not be specified dynamically. Should remapping of data be required, an explicit assignment statement may be used; e.g., to change the data in an array B from skewed to straight storage an assignment statement

```
A ← B
```

is used, where A has been declared to be a straight array.

A set is an ordered collection of elements each of which is an ordered n-tuple $(1 \leq n \leq 7)$ and the set is said to be n-dimensional. A set declaration must specify one of the attributes INCSET, MONOSET, GENSET or PATSET according as the set elements are to form an arithmetic progression (increment set), a strictly monotonic sequence (monotonic set), an arbitrary sequence (general set) or are multidimensional (pattern set), respectively, and in the latter case must also specify the size of n $(n>1)$. The declarations also may specify bounds for the integer values of the components of the n-tuple set elements, in ways analagous to the specification of array subscript bounds in ALGOL and FORTRAN, and an upper bound for the number of elements in the set. Some examples of set declarations are

```
INCSET    JJ
MONOSET   II [27, 6], KK [ 75,  75]
GENSET    A [150, 100]
PATSET    P (3) [0:20, 0:20, -5:15, 10]
```

The monotonic set II is to have at most 6 one-dimensional elements the integer values of which are to lie in the range [1, 27]. The three-dimensional pattern set P is to have at most 10 3-tuple elements the first two components of which will lie in the range [0, 20] and the third components will be in [-5, 15].

## 2.2 Expressions

Expressions which can be formed in ALGOL can, in general, also be formed in TRANQUIL. In addition arithmetic, logical and relational operators, and function designators may be used on arrays. The meaning of an operator is determined by its corresponding operands, which may be simple variables or arrays. All meaningful combinations of operands and attributes are valid; e.g., if A and B are matrices then A/B will mean $A \times B^{-1}$ if B has an inverse and the dimensions are correct. The result of a relational operator operating on two matrix operands is reduced to a single logical value through use of the qualifiers ANY and ALL, e.g.

ANY A < B or ALL A < B

Examples of set definitional expressions are

```
SET1 ← [   ]
SET2 ← [1, 2, 3, 4, 5, 6, 7, 8]
SET3 ← [1, 2, ..., 8]
SET4 ← [-2, P, Q, 25]
SET5 ← [[10, 10], [9, 8], [8, 6]]
```

where SET2 and SET3 are equivalent definitions and SET5 is a 2-dimensional pattern set. Replication factors may be used in general sets. For example:

SET6 ← [1(3), 4, 5(2)]

is equivalent to

SET6 ← [1, 1, 1, 4, 5, 5]

A useful device for the generation of a set is the run-time comparison of data values in parallel. For example, if A and B are vectors stored across PE memory,

A = {-1, 3, 2, 10} and B = {2, -3, 1, 12},

then the operation

SET7 ← SET [I : A[I] < B[I]]

where I takes on the values 1, 2, 3, and 4 simultaneously, generates the set [1, 4], the order being defined as monotonically increasing. These definitions are readily extendable to multidimensional pattern sets which are generally used for picking scattered values in an array for simultaneous operation.

The set operators INTERSECT, UNION, and COMPLEMENT (a binary operation equivalent to the relative complement) may be used in TRANQUIL and always result in the generation of a monotonic set by

reordering elements if necessary. The two additional set operators CONCAT and DELETE do not result in reordering of the elements. Some examples of set operations are:

if R = [1, 2, 3, 4, 5]
S = [2, 4, 6, 8, 10]
T = [6, 4, 6, 5, 6, 7]
U = [100, 40, 0, 13]

then R UNION U is [0, 1, 2, 3, 4, 5, 13, 40, 100]
R CONCAT S is [1, 2, 3, 4, 5, 2, 4, 6, 8, 10]
T COMPLEMENT R is [6, 7]
T DELETE R is [6, 6, 6, 7]

Finally, it is possible to create sets with multidimensional elements out of sets with scalar elements through use of the pair (,) and cartesian product (×) set operators, e.g.

[1, 2, 3, 4] , [2, 4, 6, 8] is [[1, 2], [2, 4], [3, 6], [4, 8]]
[1, 2] × [3, 4] is [[1, 3], [1, 4], [2, 3], [2, 4]]
[1, 2] × [3, 4] , [5, 6] is [[1, 3, 5], [1, 4, 6], [2, 3, 5], [2, 4, 6]]

where , has higher precedence than ×.

## 2.3 Control Statements

Control statements in TRANQUIL are used to designate sequential loops, simultaneous statement execution, and the usual conditional and unconditional transfers of control. Index sets play an integral part in these control statements. They are used as a source of values for iterative or loop control, and as a means of simultaneously specifying a number of array elements. Their association with the enablement and disablement of PE's should be obvious.

### 2.3.1 Sequential Control: The SEQ Statement

Sequential control refers to the existence of a loop through which designated index variables take on the values of a set or sets, one element at a time. It is written in the following general form:

$$\underline{\text{FOR}}\ (I_1, \ldots, I_n)\ \ \underline{\text{SEQ}}\ \ (II_1\ \{\times \mid ,\}\ \ldots\ \{\times \mid ,\}\ \ II_n)$$

$$\{\langle\text{empty}\rangle \mid\ \ \underline{\text{WHILE}}\ \langle \text{boolean expression} \rangle\}\ \ \underline{\text{DO}}\ S$$

where { } means one of the alternatives separated by | , the scope is the statement S, n is an integer, $I_i (i = 1, \ldots, n)$ are control variables, and $II_i (i = 1, \ldots, n)$ are 1-dimensional set identifiers or literal set definitions. The use of this statement is illustrated by the following examples.

(a) FOR (I, J) SEQ ([1, 2, ..., 10], [5, 10, ..., 50]) DO

A[I] ← B[I + 1] + C[J]

is evaluated as

A[1] ← B[2] + C[5];

A[2] ← B[3] + C[10];

. . . . .

A[10] ← B[11] + C[50]

Note that the comma between the two set definitions denotes pairwise ordering for the control variables values.

(b) FOR (I) SEQ ([2, 4, 6]) WHILE I < A[I] DO

A[I] ← B[I] - A[I]

will continue looping until the boolean expression is FALSE or the index set has been exhausted. As in ALGOL, no pass through the loop is made if the value of the boolean expression is FALSE after the index variable is assigned the initial value of 2.

(c) FOR (I, J) SEQ ([1, 2, 3, 4], [5, 6]) DO

B[I, J] ← A[I, J]

In this case the difference in size of the two defined sets is resolved by considering only the pairs (1, 5) and (2, 6), that is, the exhaustion of the smallest index set signals the end of the loop. To indicate otherwise an asterisk is placed after the set the exhaustion of whose elements is to be used as the stopping condition.

This means that any other sets which run out of elements before completion will be repeatedly used as many times as necessary. If the previous statement is rewritten as

```
FOR (I, J) SEQ ([1, 2, 3, 4]*, [5, 6]) DO
    B[I, J] ← A[I, J]
```

the result is

```
B[1, 5] ← A[1, 5];
B[2, 6] ← A[2, 6];
B[3, 5] ← A[3, 5];
B[4, 6] ← A[4, 6];
```

(d)
```
FOR (I, J) SEQ ([1, 2] × [6, 7, 8]) DO
    A[I, J] ← B[J, I];
```

yields

```
A[1, 6] ← B[6, 1];
A[1, 7] ← B[7, 1];
A[1, 8] ← B[8, 1];
A[2, 6] ← B[6, 2];
A[2, 7] ← B[7, 2];
A[2, 8] ← B[8, 2];
```

where the lengths of the two sets do not create the problem that occurred with the pairwise operator. This example also illustrates that the frequency of element change is greatest for the rightmost set used.

2.3.2 Simultaneous Control: The SIM Function and the SIM Statement

The parallel structure of ILLIAC IV is utilized in TRANQUIL by the specification of simultaneous control functions and statements. The general form of the SIM function is:

SIM BEGIN <assignment statement>; ...; <assignment statement>END

where the enclosed assignment statements are executed simultaneously, i.e., the data used by any one of them is the data available before the SIM function was encountered.

The general form of the SIM statement for simultaneous control is:

FOR ($I_1$, ..., $I_n$) SIM ($II_1$ {× | ,} ... {× | ,} $II_m$) DO S

where m, n are integers, $I_i$ (i = 1, ..., n) are control variables, $II_i$ (i = 1, ..., m) are k-dimensional sets ($0 < k \leq 7$), n equals the total number of dimensions of all $II_i$, and S is a statement. For this statement each substatement $S_i$ of S is executed with the data available before it is reached, i.e., just as if a SIM function was placed around each $S_i$. In this regard it is important to note that simultaneous control is not loop control, but designates that each $S_i$ is to be executed in parallel and thus the order of the associated sets is not important. Some examples of the use of SIM are:

(a) FOR (I, J) SIM ([1, 2, 3]*, [4, 5]) DO

A[I, J] ← B[J, I]

is evaluated as

```
SIM BEGIN  A[1, 4] ← B[4, 1];
           A[2, 5] ← B[5, 2];
           A[3, 4] ← B[4, 3]
    END
```

```
(b) FOR (I, J) SIM (II × JJ) DO
      BEGIN
        C[I, J] ← 0;
        FOR (K) SEQ (KK) DO
          C[I, J] ← C[I, J] + A[I, K] × B[K, J]
      END
```

is a general routine for the multiplication of two compatible matrices A and B if the index sets II and JJ specify the rows of A and the columns of B, respectively.

It should be noted that when a set is used in a sequential or simultaneous control statement, it cannot be altered within the scope of that statement.

### 2.3.3 Nested SEQ and SIM Statements

The SEQ and SIM control statements described above may be nested. The effect of nesting is clear except when a SIM statement occurs within the scope of another SIM statement, in which case

statements inside the scope of both are executed under the control of sets which are related by the cross product operator, for example:

```
FOR (I, J) SIM (II, JJ) DO
   BEGIN FOR (K) SIM (KK) DO
         BEGIN
            Area A
         END;
      . . .
END
```

where the control statement in effect in area A is, in effect,

FOR (I, J, K) SIM (II, JJ × KK) DO

2.3.4 If Clauses

General forms:

(a) IFSET <indexed boolean expression> THEN

(b) IF { FOR ($I_1$, ..., $I_n$) SIM ($II_1$ {×|,} ... {×|,} $II_m$)| <empty>}
{ANY|ALL|<empty>} <boolean expression> THEN

If clauses may be used in arithmetic, boolean, set and designational expressions. The boolean expression in form (a) must involve a control variable under SIM control and thus not have a single logical value. This is meaningful in arithmetic and boolean expressions of assignment statements having left parts which use the

same control variable, and also in conditional statements where the control variable is used in the left part of some associated assignment statement. An example of this use is:

```
FOR (I) SIM ([1, 2, ..., 100]) DO
   T[I] ← IFSET A[I] < B[I] THEN A[I] ELSE B[I]
```

is equivalent to

```
FOR (I) SIM ([1, 2, ..., 100]) DO
   IFSET A[I] < B[I] THEN T[I] ← A[I]
                     ELSE T[I] ← B[I]
```

In either form T[I] ← A[I] for all values of I for which the value of the boolean expression is TRUE and T[I] ← B[I] otherwise.

The form (b) results in only a single boolean value based on the ANY or ALL modifier, and the scope of the SIM control (if explicitly present) extends over the boolean expression only. If the vector A of length 2 has elements 5 and 10, the if clause test

```
IF FOR (I) SIM ([1, 2]) ANY A[I] < 7
```

has the value true since A[1] < 7. The same result is achieved by use of the if clause

```
IF ANY A < 7
```

# 3. THE TRANQUIL COMPILER AND ITS IMPLEMENTATION

## 3.1 Introduction

The syntax of TRANQUIL has been specified in a form which is accepted by the syntax preprocessor of the Translator Writing System (TWS) [5,6,7,8] being built at the University of Illinois. The preprocessor automatically generates the parsing algorithm for the compiler. In pass 1 of the compiler the recognition of source code constructs invokes calls, via the action numbers embedded in the syntax definition, to semantic actions. These actions build descriptor tables containing, in part, information about declaration types, attributes, and block structure, and transform the source code into an intermediate language form which is composed of operators and operands, the latter being references to the descriptor tables.

Pass 2 is the main body of the compiler. The intermediate language stream is read and operators call pass 2 semantic actions (on the basis of their context) for generation of assembly language instructions using associated operands. A number of other important considerations arise, among which are the storage of arrays and sets, and the efficient allocation and use of CU registers. These problems and some solutions are discussed in the following sections.

## 3.2 Array Blocking and Storage Allocation

The compiler partitions all arrays into 2-dimensional blocks the maximum size of which is $64q \times 64q$ words ($q = 1, 2,$ or $4$ according as 1, 2 or 4 quadrants of 64 PE's are being used) since

ILLIAC IV may be regarded as an array with 2048 rows (number of words in a PE) × 64q columns (number of PE's). For the purposes of this section it will be assumed that q = 1. The sizes of the blocks obtained by the array partitioning then fall into 4 categories:

(a) $64 \times 64$

(b) $m \times 64$

(c) $64 \times n$

(d) $m \times n$ $\quad m, n < 64$

which are called SQUARE, HBLOCK, VBLOCK and SBLOCK, respectively. Small blocks belonging to the same array are packed together to form a larger block, details of which are given by Muraoka[9]. Figure 2 illustrates the partitioning of a 3-dimensional array into 12 blocks and Figure 3 illustrates how the smaller subblocks are packed together to form larger blocks. The partitioning of an array into blocks is independent of the mapping function; i.e., for a SKEWED array skewing is done after partitioning.

All array operations and data transfers between ILLIAC IV disk and PE memory are done in terms of these blocks. A block of size m × n may be placed in any m adjacent words (rows) stored in n adjacent PE's in PE memory. Blocking facilitates the use of arrays which are larger than the PE memory. All data is normally stored on the $10^9$ bit ILLIAC IV disk and blocks are only brought into PE memory (at a transfer rate of $.5 \times 10^9$ bits/second) as

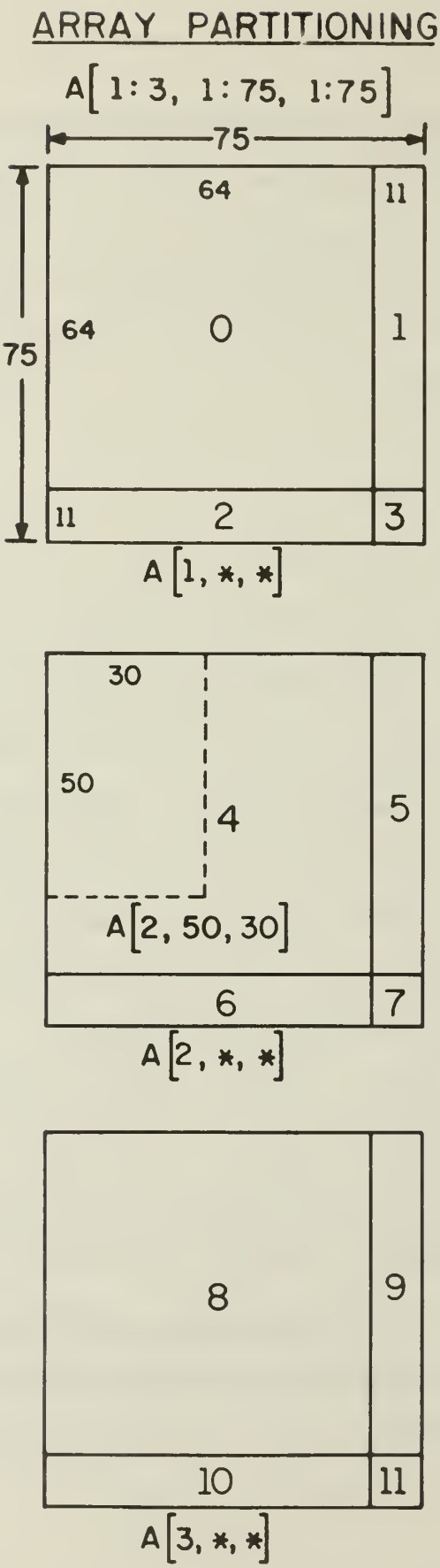


Figure 2. The partitioning of array A[1:3, 1:75, 1:75] into blocks for array storage

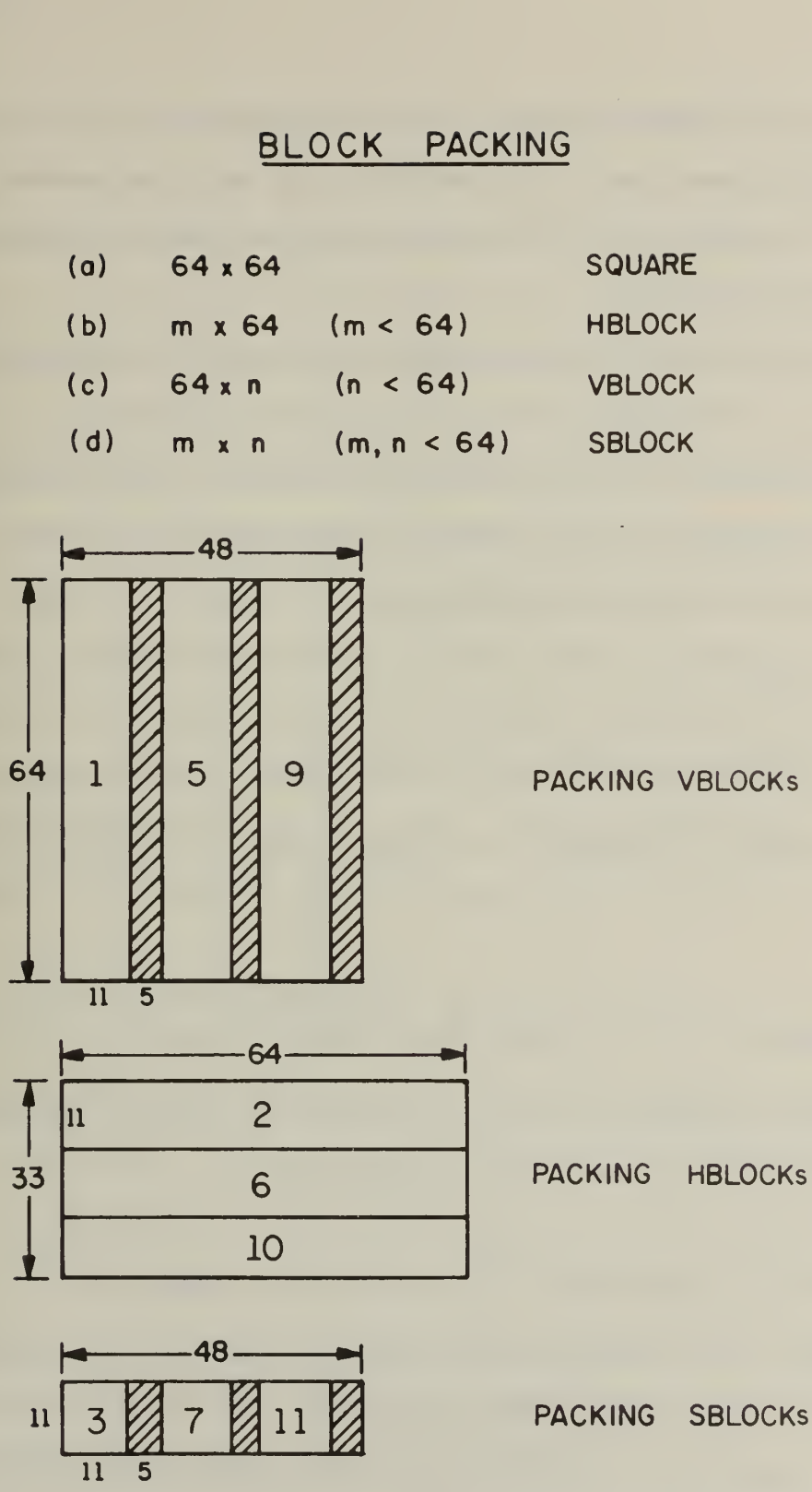


Figure 3. Block packing for array A[1:3, 1:75, 1:75]

required. The TRANQUIL compiler automatically generates block transfer I/O requests, which are handled by the operating system. Thus, it is possible to write a TRANQUIL program which includes no explicit data transfers.

The effective address of any array element is obtained by computing the block number, which determines an absolute base address (the address of the upper left-most element of the block if the block is in memory) and a relative address within that block. The block number for an element $A[i_1, i_2, \ldots, i_{n-1}, i_n]$ of an array declared

$A[\ell_1 : u_1, \ldots, \ell_n : u_n]$ is given by

$$((\ldots((i_1-\ell_1)M_2 + (i_2-\ell_2))M_3 + \ldots(i_{n-2}-\ell_{n-2}))M'_{n-1} + i''_{n-1})M'_n + i''_n,$$

where

$$M_i = u_i - \ell_i + 1,\ M'_i = (M_i + 63)\underline{\text{div}}\ 64,\ i''_k = (i_k - \ell_k)\underline{\text{div}}\ 64.$$

If the array is SKEWED the relative PE number and relative PE address of the element in the specified block are given by

$$[(i_{n-1} + i_n) - (\ell_{n-1} + \ell_n)] \bmod 64 \quad \text{and} \quad (i_{n-1} - \ell_{n-1}) \bmod 64,$$

respectively. After skewing of the blocks in Figure 2, the element A[2, 50, 30] is specified by the block number, and PE number and PE address relative to the base address of the block, which have values 4, 14 and 49, respectively. In most cases some or all of the

elements in a row or column are used simultaneously. In the case of column operation, for example, each PE can simultaneously compute the relative address (index value) which it will require.

In allocating memory space for a block a linked space list, which keeps track only of the number of rows of memory which have been used, is utilized. If a block of size $m \times 64$ is to be stored, the list is searched to locate a space of m adjacent rows and the block is stored there. In the case of a block of size $m \times n$ $(m, n < 64)$ 64 rows of PE memory may be allocated and a sublist corresponding to this $64 \times 64$ block of storage is established. The sublist consists of a boolean array in which each bit represents the use or otherwise of each $8 \times 8$ subblock of the associated $64 \times 64$ block of storage, thus allowing several small blocks to be stored together in a larger unit.

## 3.3 The Storage and Use of Sets

Associated with the introduction of sets in TRANQUIL is the task of finding storage schemes which can be used efficiently. Sets can be used for loop control, for enabling or disabling PE's, or for PE indexing. The increment set can be used in all three ways and is stored using two words per set. One word contains the first element, the increment, and the limit packed for use by CAR instructions. The other word is used as a bias value in the case where negative elements are, or may be, in the set.

When an increment set is used for sequential control, CU test and increment instructions operate on the appropriate CAR

register. When an increment set is used for simultaneous control, it can be expanded at run time into mode words or explicit numbers. Mode words are 64-bit words used to store the elements of a set by position number, where a 1 in the n-th bit indicates that n is an element of the set. These words are most frequently used in PE mode registers to enable and disable PE's. Mode words can be generated from an increment set by using a memory row that contains the PE numbers in ascending and descending order and regular mode patterns similar to the 4 PE system shown in Figure 4. In the figure the mode pattern was formed by considering the $b_0$ bits to be all ones, the $b_1$ bits to be alternating zeros and ones, the $b_2$ bits to be two zeros alternating with a one, and finally the $b_3$ bits to be three zeros alternating with a one. In the general 64 PE case, the word in the i-th PE is

| Mode pattern | i | 63-i |
|---|---|---|

where the 32-bit mode pattern results by considering the $b_0$ bits to be all ones: i.e., all PE's on when in use, the $b_1$ bits having the pattern 0101..., the $b_2$ bits 001001..., and the $b_i$ bits $0_i1 0_i 1$ where $0_i$ stands for i zeros.

Now consider the example of expanding the increment set JJ of Appendix **A** into mode words and explicit numbers. The set JJ is used simultaneously in forming the comparison set KK by a boolean test on the skewed arrays A and B. For a given I every other element of A, as signified by JJ, is moved to the A register in the PE's,

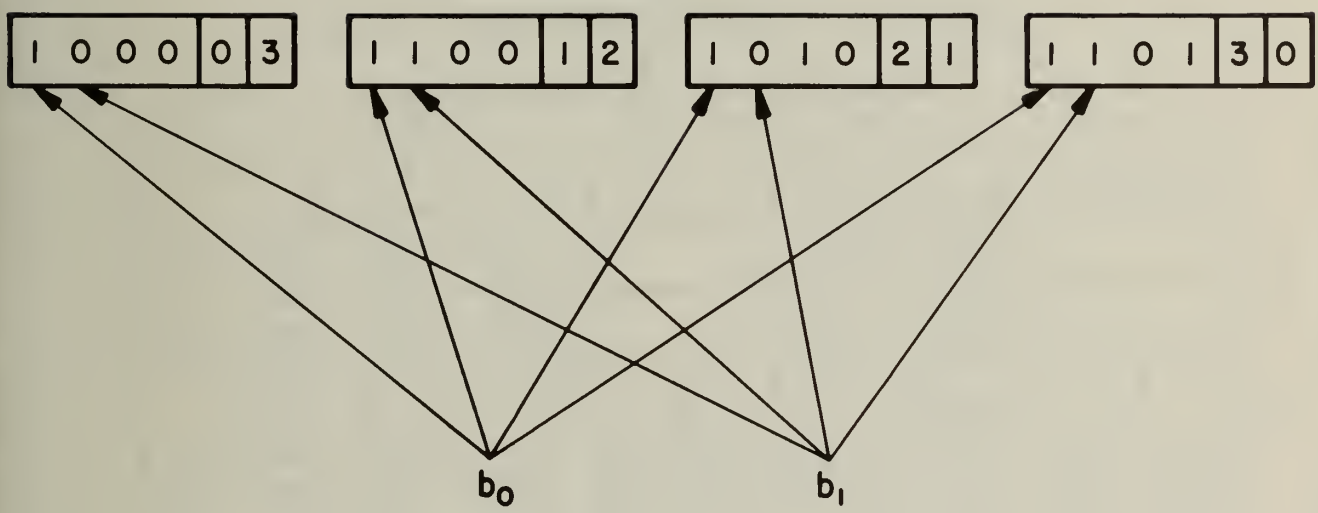


Figure 4. Mode patterns and explicit values for increment sets

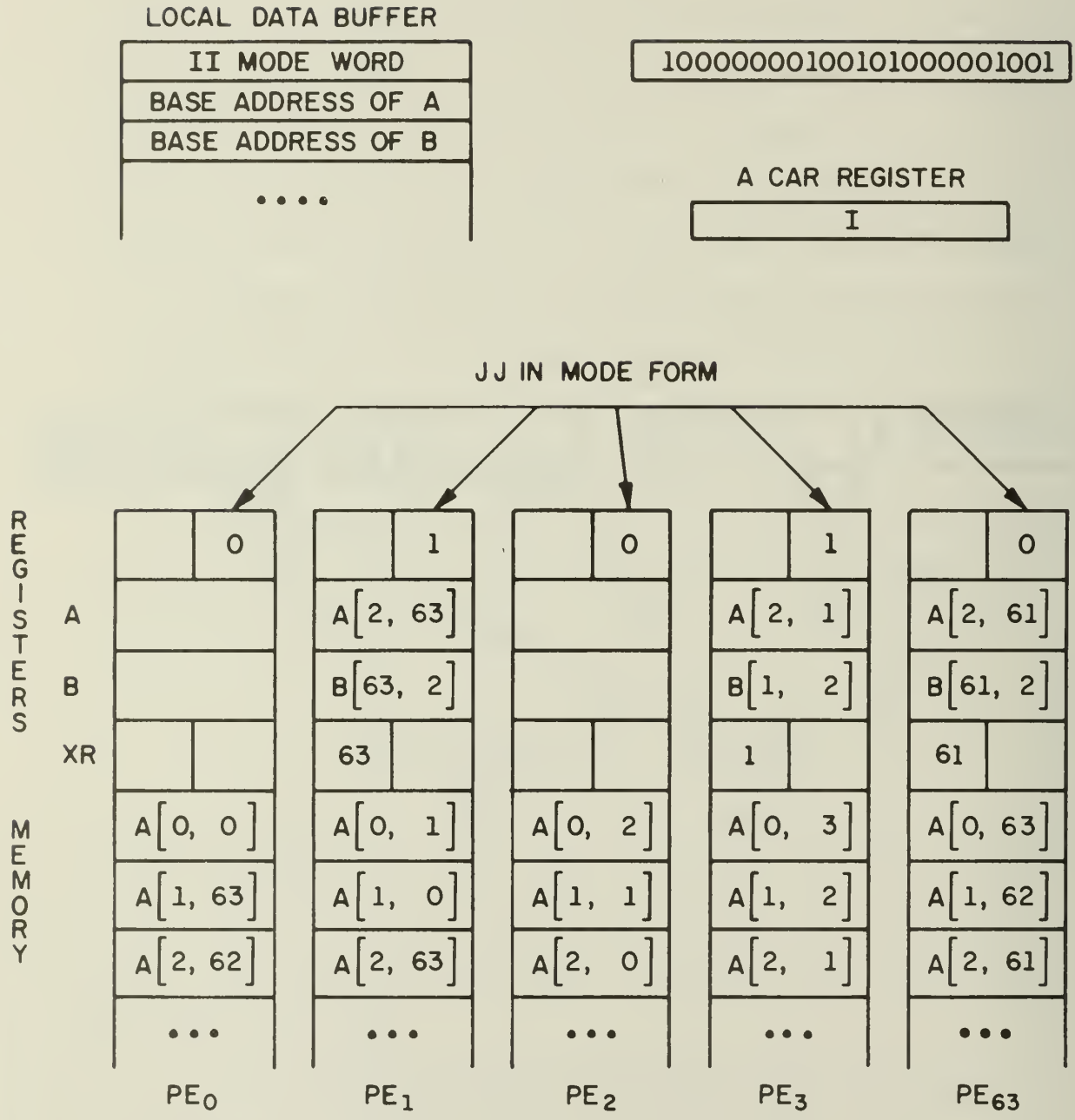


Figure 5. PE and CU status for I = 2 in the example problem

using the base address of A that has been brought to the CU data buffer as indicated in Figure 5. Every other element corresponds to the $b_1$ bit mode pattern, this pattern appearing in the PE mode positions of Figure 5 which is based on Figure 1. The case for I = 2 is shown. Every other element of the I-th column of B is moved to the B register. In this case every other ascending PE number is used in the PE index registers, XR, with appropriate routing to account for the skewed storage. For I = 2 in Figure 5, an end around route of two is necessary. Every other element of a column is fetched to the B register again by the use of JJ in mode form.

The sets II and KK in Appendix A are examples of monotonic sets and are stored in mode form. For looping on II the CU does leading ones detection on the II mode pattern, illustrated in Figure 5, to determine the explicit set elements used in the CAR as CU indexes for array A, and KK is used in mode form in the PE mode registers under simultaneous control. Monotonic sets can also be stored as explicit numbers for index use. The general set is always stored in explicit form, for obvious reasons, and pattern sets are stored as mode words.

The actual management of mode and explicit storage schemes involves the use of a permanent storage area and a stack. All set definitions are stored in the permanent storage area except comparison sets. The stack is used for storing current set values that are obtained from the permanent area or generated by the user or compiler. The stack is also used because program flow is not

known at compile time, in general, due to data based branches, and thus changes in these sets are unpredictable. In PE memory the storage of either the mode or explicit set representation begins on an 8 word boundary to make best use of the 8 word CU fetch capabilities. Further details of the handling of sets are given by Wilhelmson.[10]

## 3.4 CU Storage Allocation and Use

The allocation and use of CU registers is a very important ILLIAC IV problem since CU instructions which cannot be overlapped with PE instructions leave all PE's idle. The allocation of CU registers for needs known at compile time is accomplished by calling one of a group of procedures that have an underlying allocation priority system and that use compile-time pointers to and from important tables and storage locations. The local data buffer is divided into two parts: the lower part for use as determined at compile time and the upper part for dynamic use at run time. The lower part is further divided into three parts. The first is one word used for bit storage and later testing. A list of free bits is kept, bits being assigned on a space available basis. The second part is 16 words in length, where the use of this space is kept to low priority requests unless space is needed for high priority requests. Priorities range from 0 to 3 and are assigned by the compiler writer, these assignments being based on intuition and experience. The third part has n words ($0 \leq n \leq 47$), where the optimal value of n is yet to be determined. The space is used as much as possible for high priority requests. The reason for this device is

to try and keep low priority requests in the lower area, since the lower area will be used to store 8-word blocks transmitted to the CU. When the local data buffer becomes full a word is freed, with appropriate storage and pointer modification if necessary. Thus a user can let the procedures free words when necessary unless he wishes to do so earlier.

Three of the CAR registers are allocated using the same priorities as in the local data buffer. The fourth is a free register which may be used in a variety of ways.

Another CU problem is connected with its particular data composition, and results from transfers. For example, at the beginning of a loop data in the CU has a certain composition. This composition should be reinstated each time through the loop. This is made possible by remembering the composition of the CU data at the beginning of the loop.

For backward transfers code to set up the CU properly is placed at the jump point while for forward transfer code is placed at the location transferred to. The priority scheme may appear to add to the problem of moving words in and out of CU memory at transfer points. This scheme has been developed since 8-word block stores to PE memory are not allowed; only one word at a time can be stored in PE memory by the CU.

## 3.5 Assignment Statements

The use of sets, the notion of SIM, the number of different types of arithmetic and storage schemes, combined with the need to

compile efficient code for a parallel machine necessitate a substantial analysis of each assignment statement. We now consider this analysis as it is carried out in pass 2 of the compiler.

The analysis is effected in several passes over the postfix intermediate language. Consider the last assignment statement in the example in Appendix A:

$$A[I, K] \leftarrow A[I + 1, K] + B[I, K + 1];$$

Before we even begin to generate code a decision must be made as to which index is to be processed simultaneously (i.e., across the PE's) and which is to be done sequentially. The first pass over the intermediate language determines this and also copies the intermediate language into a table to be used for future passes. When a set linked* identifier is entered in the table, additional information provided by the set definition or declaration is also entered. In the case of I, which is linked via SIM control to II, the set is known exactly and precise information from the set definition is entered in the table. For K the compiler makes an estimate of the size and density of the set based on the upper bounds given in the declaration of KK.

---

*We say that I is set linked to II in a statement like FOR (I) SIM (II) DO.

---

In general, when operations are performed on pairs of subscripts or pairs of subscripted arrays, information about the interaction between these subscripts must be generated. For example, in

the case of the subscript expression I + 1 in the example above, the addition of 1 in no way alters the size, density, or type of the set. Thus, the information provided for I will be recopied with the + operand.

After the subscript expression has been processed, a check is made to see how well the type of set resulting from the index expression will work with the particular dimension of the array involved. In the example there are only two-dimensional skewed arrays in which either columns or rows can be easily accessed in parallel. If one of the arrays were straight, then at this point it would be discovered that no set will work well for the column index, because each column is stored in a single PE. This information plus information about the set density, set size, and the array size are all combined to compute a probable efficiency; i.e., the number of PE's that will probably be enabled if this index were varied simultaneously. Of course, it is easy to think up cases in which the estimate will be totally wrong, but in most practical cases encountered, the estimate is reasonable. A table of these probable efficiencies is generated for each set. If the set appears in different subscripts, then on the second occurrence the new estimate is set to the minimum of the previous and present estimates.

When the end of the assignment statement is reached, the table of probable efficiencies is sorted and the result of this determines the order in which the indexes will vary. In the example K will be the index chosen to vary across the PE's because the set II

is known to be small (6 elements) and the declaration of KK holds the probability of it being fairly large. Now an outer loop must be compiled to generate sequentially the elements of II. Finally, the remainder of the statement is compiled. The effect of the code that is compiled for the example assignment statement follows.

One local data buffer location is set aside as an index to the mode words of KK. Four more locations are set aside for the base addresses of the subblocks of the arrays A and B. The first mode word for KK is loaded and the leading ones detector is used to set the first value of K. This value, plus the base address of A, plus 1, plus the index set 0, 1, 2, ..., 63 in the PE index registers is used to access the first column of A. In a similar manner the address for the first row of B is fetched, loaded into RGR and a route left one PE is performed. The addition is executed and the first mode word for KK is used to store the result in A. Now the same process is repeated for the next subblock of A, except that the mode pattern for KK must be ended with a word having 11 1's followed by 0's, because the second subblock of A is only 11 words wide. Additional complications, such as pairwise SIM control specification, small subarrays, and SIM blocks add to the complexity, but not to the substance, of the algorithm outlined above.

It is clearly impossible to efficiently compile a single short assignment statement for ILLIAC IV, but it is conceivable that a large number of simple assignment statements could be integrated into a fairly efficient ILLIAC IV program. Incorporating such a

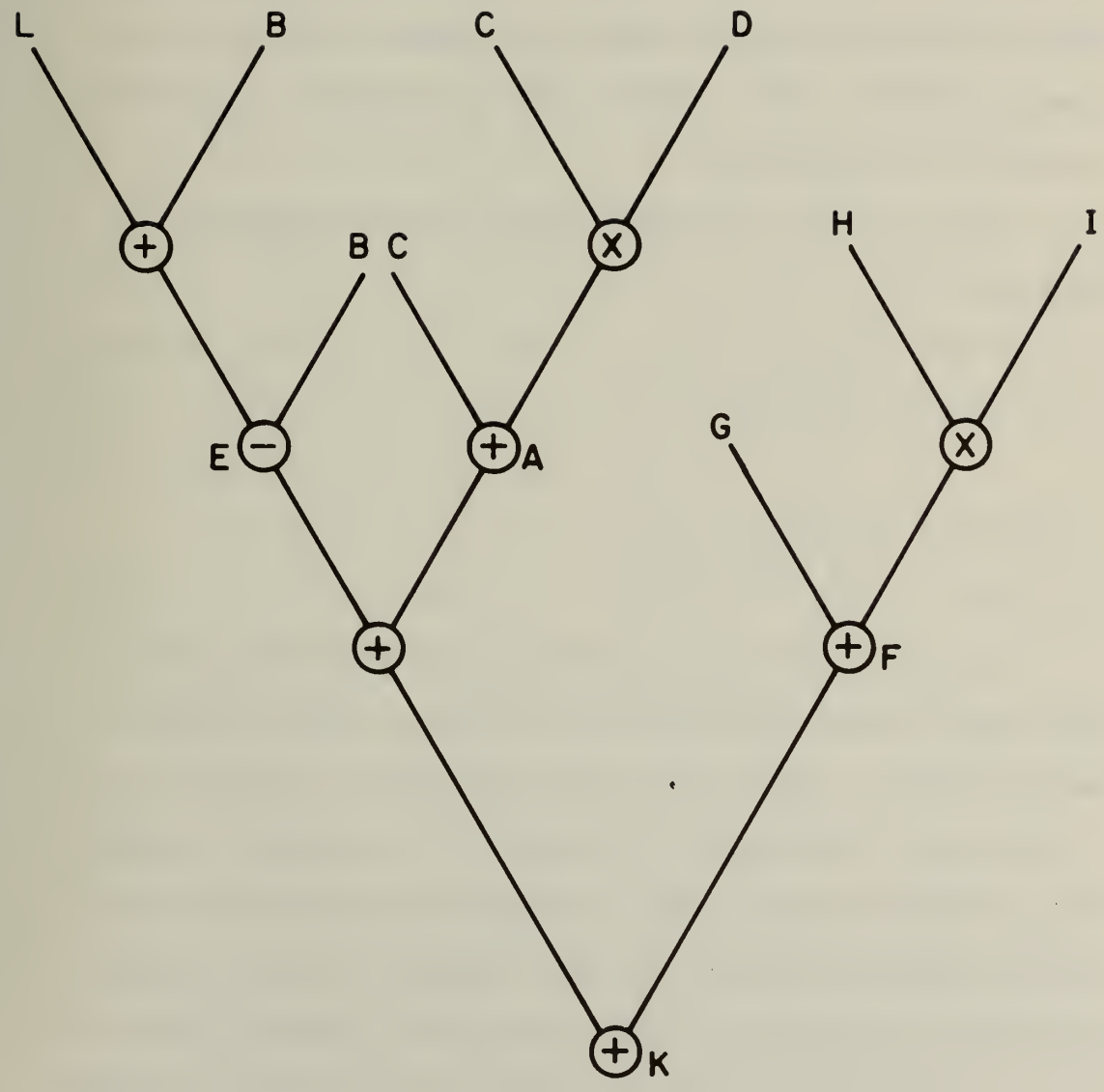


Figure 6. The tree structure for a set of interrelated arithmetic statements

feature into a compiler presents two basic problems. The first is an algorithm for efficiently integrating a large number of interrelated assignment statements. Ordinarily the simple assignment statements will be scattered throughout the program. Also, many of the sequential calculations that are prime targets for an integration scheme are likely to be embedded as subexpressions in assignment statements containing SIM controlled variables. Filtering out and gathering together these candidates for the integration scheme constitutes the second problem.

Figure 6 is a tree structure for the set of assignment statements:

$$A \leftarrow B + C \times D$$

$$E \leftarrow L + B - C$$

$$F \leftarrow G + H \times I$$

$$K \leftarrow A + E + F$$

No node on this tree can be calculated until all nodes on subbranches have been calculated. The method of computing such a tree on ILLIAC IV involves first mapping assignment statements into PE's, in a more or less arbitrary manner. The assignment statements are restricted to a small number of operations like addition, subtraction, multiplication and division. ILLIAC IV can only perform one of these operations at a time. A count of the number of PE's that can take advantage of each of these operations is made and that operation which will be executed by the most PE's is the one that

code is compiled for. Then the PE counts for all operations are revised and the process continues until all calculations have been performed. A similar algorithm is used to do routing to bring the results computed in one PE to the PE's where they are needed. This algorithm is invoked whenever the number of PE's eligible for any operation falls below a certain limit.

The problem of gathering together assignment statements for processing by this method is many faceted. What is desired is a rearrangement of the program where simple assignment statements, simple subexpressions, and simple expressions generated by the compiler, like address calculation, have been brought together at several collection points. To rearrange code in this manner requires an extensive analysis of the overall program to determine what subexpressions and statements can be moved, and how far.

This analysis is carried out at the intermediate language level. The collection points are determined to be at the beginning of blocks, subexpressions are moved as physically high up in the code as possible, except that they are not moved past a block head unless they can be moved to the head of an outer block. The method produces a number of bonuses. Calculations inside loops tend to be moved outside when logically permissible. Thus, it is profitable to move nonsimple subexpressions also. Further, duplicate subexpressions can easily be eliminated because they tend to gather at the same point. Finally, for each block a record is made of what variables are nondynamic within that block. Thus, in pass 2, any expressions generated

using these variables can be added to the collection of subexpressions at the beginning of the appropriate block. At the head of this block, a transfer to the end of the block is compiled, and when all code in the body of the block has been generated, the complete collection of assignment statements is compiled followed by a transfer back to the beginning of the block. More details of this type of analysis are given by Budnik.[11]

## 4. SUMMARY

Designing and implementing a language and its compiler for ILLIAC IV presents a number of problems not encountered with procedure oriented languages for sequential machines. In the design of the language these problems have been met, primarily through the use of sets as indexes and the introduction of language elements for explicit denotation of simultaneous operations. Experience has shown that the resulting notation is as easy to learn as that of conventional languages and in most instances it is more concise.

The task of efficiently compiling a language such as TRANQUIL for the ILLIAC IV is more difficult than compiling for conventional machines, simply because the standard compiling techniques are inadequate, thus requiring new compilation algorithms to be invented. These techniques will undoubtedly be refined as further experience is gained with the use of ILLIAC IV and parallel languages. However, the completion of the major parts of the TRANQUIL compiler has already demonstrated that reasonably efficient object code can

be generated for a large class of array-type problems which have been programmed in TRANQUIL.

Several features of TRANQUIL have been omitted from this paper, notably input/output statements and procedures. Execution time input/output will be from/to the ILLIAC IV disk (secondary storage) in blocks of data. Most of these data transfers will be implicitly specified in TRANQUIL programs. However, some explicit specification of unformatted data transfers will be provided. The provision of additional software to facilitate format specified transfer of data between external peripherals (tertiary storage) and the ILLIAC IV disk is planned. The specification of procedure declarations, and their use in a parallel environment, is under investigation. Additional features being considered for later incorporation into the TRANQUIL compiler include overlayable code segments, quadrant configuration independent code, and more specialized data structures and mapping functions. Although aspects of the language and its compiler are still being developed, it has been demonstrated that TRANQUIL is a highly satisfactory and useful complement to the ILLIAC IV hardware system design.

ACKNOWLEDGEMENTS

The research reported in this paper was supported in part by the Department of Computer Science, University of Illinois at Urbana-Champaign and in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center, under Contract No. USAF 30(602)4144.

REFERENCES

1 G BARNES R BROWN M KATO D KUCK D SLOTNICK R STOKES

The ILLIAC IV computer

IEEE Transactions on computers vol C-17 pp 746-757 August 1968

2 D J KUCK

ILLIAC IV software and applications programming

IEEE Transactions on computers vol C-17 pp 758-770 August 1968

3 M B WELLS

Aspects of language design for combinatorial computing

IEEE Transactions on computers vol EC-13 pp 431-438 August 1964

4 K E IVERSON

A programming language

John Wiley New York 1962

5 R S NORTHCOTE

The structure and use of a compiler-compiler system

Proceedings of the 3rd Australian computer conference 1966

6 F L DEREMER

On the generation of parsers for BNF grammars: an algorithm

Proceedings of the spring joint computer conference 1969

7 A J BEALS

The generation of a deterministic parsing algorithm

Report No 304 Department of Computer Science University of Illinois at Urbana 1969

8 H R G TROUT

A BNF like language for the description of syntax directed compilers

Report No 300 Department of Computer Science University of Illinois at Urbana 1969

9 Y MURAOKA

Storage allocation algorithms in the TRANQUIL compiler

Report No 297 Department of Computer Science University of Illinois at Urbana 1969

10 R B WILHELMSON

Control statement syntax and semantics of a language for parallel processors

Report No 298 Department of Computer Science University of Illinois at Urbana 1969

11 P P BUDNIK

TRANQUIL arithmetic

Report No 296 Department of Computer Science University of Illinois at Urbana 1969

## APPENDIX A: A SAMPLE TRANQUIL PROGRAM

```
BEGIN
REAL  SKEWED  ARRAY A, B[1:75, 1:75];
INCSET JJ;
MONOSET  II(1) [27, 6], KK(1) [75, 75];
INTEGER  I, J, K;
II  ←  [2, 10, 13, 15, 21, 24];
JJ  ←  [2, 4, . . ., 74];
FOR (I)  SEQ  (II)  DO
      BEGIN FOR (J)  SIM (JJ)  DO
                KK ← SET (J:A[I,J] < B[J,I]);
            FOR (K)  SIM (KK)  DO
                A[I,K] ← A[I+1,K] + B[I,K+1]
      END;
FOR  (I,K)  SIM (II × KK) DO
            A[I,K] ← A[I+1,K] + B[I,K+1]
END
```

# APPENDIX B: A SPECIFICATION OF THE SYNTAX OF TRANQUIL

A brief description of the syntax metalanguage is given in APPENDIX C.

## B.1 Program

```
< PROGRAM > ::= < BLOCK >

< BLOCK > ::= BEGIN  list [< DECLARATION > #;]
                  list < STATEMENT > separator #;
                  [#;]&  END;

< STATEMENT > ::= < NONEMPTY STATEMENT > | < > ;

< NONEMPTY STATEMENT > ::= [ < * I > : ]*
                  [ < CONTROL STATEMENT > |
                  GO TO < DESIGNATIONAL EXPRESSION > |
                  BEGIN < NONEMPTY STATEMENT > [# ; < STATEMENT >]* END
                  < BLOCK > |
                  < IF CLAUSE > < STATEMENT > [ELSE < STATEMENT > ] & |
                  < ASSIGNMENT STATEMENT > ] ;
```

## B.2 Declarations

```
< DECLARATION > ::= < VARIABLE DECLARATION > |
                  < ARRAY DECLARATION > |
                  < PEM RESERVE DECLARATION > |
                  < PEM ASSIGNMENT DECLARATION > |
                  < SET DECLARATION > |
                  < SWITCH DECLARATION > |
                  < LABEL DECLARATION >;
```

B.2.1 Variable Declarations

< VARIABLE DECLARATION > ::= < ATTRIBUTE >
[ COMPLEX ]& list < *I >
separator , ;

< ATTRIBUTE > ::= BOOLEAN | REAL | REALS | REALD | INTEGER |
INTEGERS | INTEGERL | BYTE8 | BYTE16 ;

B.2.2 Array Declarations

< ARRAY DECLARATION > ::= [ < ATTRIBUTE > ]& [ < MAPPING FUNCTION > ]&
ARRAY < ARRAY LIST > | [ < ATTRIBUTE > ]& ARRAY
( < PEM AREA > ) < ARRAY LIST > ;

< MAPPING FUNCTION > ::= STRAIGHT | SKEWED | SKEWED PACKED | CHECKER ;

< ARRAY LIST > ::= list [ list < *I > separator , < BOUND LIST > ]
separator , ;

< BOUND LIST > ::= #[ [ list [ [ * | ** | # | ## ]&
< ARITHMETIC EXPRESSION > : < ARITHMETIC EXPRESSION > ]
separator , | list [ [ * | ** | # | ## ]&
< ARITHMETIC EXPRESSION > ] separator , ] #] ;

< PEM AREA > ::= < *I > ;

B.2.3 PEM Reserve Declarations

< PEM RESERVE DECLARATION > ::= PEMEMORY < PEM AREA NAME >
#[ < UNSIGNED INTEGER >,< UNSIGNED INTEGER > #];

< PEM AREA NAME > ::= < * I > ;

< UNSIGNED INTEGER > ::= < * N > ;

B.2.4 PEM Assignment Declarations

< PEM ASSIGNMENT DECLARATION > ::=

PEM [ < PEM ASSIGNMENT CONSTRUCT > |

BEGIN list < PEM ASSIGNMENT CONSTRUCT >

separator ; END ] ;

< PEM ASSIGNMENT CONSTRUCT > ::=

< PEM ASSIGNMENT STATEMENT > |

< SET ASSIGNMENT STATEMENT > | < PEM FOR STATEMENT > ;

< PEM ASSIGNMENT STATEMENT > ::= < * I >

#[ [ < UNSIGNED INTEGER > | < * I > ] ,

[ < UNSIGNED INTEGER > | < * I > ] #] ← < * I >

#[ list < ARITHMETIC EXPRESSION > separator , #] ;

< PEM FOR STATEMENT > ::=

FOR ( < SET VARIABLE LIST > ) SIM

( < SET NAME LIST > ) DO

[ < PEM ASSIGNMENT CONSTRUCT > |

BEGIN list < PEM ASSIGNMENT CONSTRUCT >

separator ; END ] ;

B.2.5 Set Declarations

< SET DECLARATION > ::= [ INCSET | MONOSET | GENSET | PATSET ] list

< SET SEGMENT > separator , ;

< SET SEGMENT > ::= list < * I > separator ,

[ (< * N >) ]& [ #[ list [ < ARITHMETIC EXPRESSION >

```
[: < ARITHMETIC EXPRESSION > ]& ]
   separator , #] ]& ;
```

B.2.6 Label and Switch Declarations

```
< LABEL DECLARATION > ::= LABEL  list < * I > separator , ;
< SWITCH DECLARATION > ::= SWITCH < * I > ← list
                < DESIGNATIONAL EXPRESSION > separator , ;
```

B.3.1 Control Statements

```
< CONTROL STATEMENT > ::= FOR (< SET VARIABLE LIST >)
                [SEQ | SIM] (< SET NAME LIST >) DO < STATEMENT > |
                FOR (< SET VARIABLE LIST >) SEQ (< SET NAME LIST >)
                WHILE < BOOLEAN EXPRESSION > DO
                < STATEMENT > | < SIM BLOCK > ;
< SET VARIABLE LIST > ::= list < * I > separator , ;
< SET NAME LIST > ::= list [ < * I > [ #* ]&
                | < SET DEFINITION TAIL > ] separator [ , | × ] ;
< SIM BLOCK > ::= SIM BEGIN  list < ASSIGNMENT STATEMENT >
                separator #; [ #; ]& END ;
```

B.3.2 Set Definitions

```
< SET DEFINITION TAIL > ::= #[ < LIST SET > #] |
                < COMPARISON SET > ;
< LIST SET > ::= < ELEMENT > [, < ELEMENT >,...,< ELEMENT > |
                , < ELEMENT > ]* | < > ;
```

```
< ELEMENT > ::= #[ list < ARITHMETIC EXPRESSION > separator , #] |
                < ARITHMETIC EXPRESSION >
                [ (< ARITHMETIC EXPRESSION >) ]& ;
< COMPARISON SET > ::= SET #[ list < * I > separator , :
                < BOOLEAN EXPRESSION > #] ;
```

B.4 Designational Expressions

```
< DESIGNATIONAL EXPRESSION > ::= < SIMPLE DESIGNATIONAL EXPRESSION > |
                < IF CLAUSE > < SIMPLE DESIGNATIONAL EXPRESSION >
                ELSE < DESIGNATIONAL EXPRESSION > ;
< SIMPLE DESIGNATIONAL EXPRESSION > ::= (< DESIGNATIONAL EXPRESSION >) |
                < SWITCH IDENTIFIER > #[ < ARITHMETIC EXPRESSION > #] |
                < LABEL IDENTIFIER > ;
< SWITCH IDENTIFIER > ::= < * I > ;
< LABEL IDENTIFIER > ::= < * I > ;
```

B.5 IF Clauses

```
< IF CLAUSE > ::= IF [ < CONTROL HEAD > ]& [ ANY | ALL ]&
                < BOOLEAN EXPRESSION > THEN |
                IFSET < BOOLEAN EXPRESSION > THEN ;
< CONTROL HEAD > ::= FOR (< SET VARIABLE LIST >) SIM (< SET NAME LIST >) ;
```

B.6 Assignment Statements

```
< ASSIGNMENT STATEMENT > ::=
                < BOOLEAN ASSIGNMENT STATEMENT > |
                < ARITHMETIC ASSIGNMENT STATEMENT > |
                < SET ASSIGNMENT STATEMENT > ;
```

B.6.1 Boolean Assignment Statements

< BOOLEAN ASSIGNMENT STATEMENT > ::= list [ < * I > ← ]
< BOOLEAN EXPRESSION > ;

< BOOLEAN EXPRESSION > ::= < SIMPLE BOOLEAN > | < IF CLAUSE >
< SIMPLE BOOLEAN > ELSE < SIMPLE BOOLEAN > ;

< SIMPLE BOOLEAN > ::= < BOOLEAN FACTOR > [ [ OR | IMP | EQV ]
< BOOLEAN FACTOR > ] * ;

< BOOLEAN FACTOR > ::= < BOOLEAN SECONDARY >
[ AND < BOOLEAN SECONDARY > ] * ;

< BOOLEAN SECONDARY > ::= < BOOLEAN PRIMARY > | NOT < BOOLEAN PRIMARY > ;

< BOOLEAN PRIMARY > ::= TRUE | FALSE | < * I > | < RELATION > |
( < BOOLEAN EXPRESSION > ) ;

< RELATION > ::= < ARITHMETIC EXPRESSION > ELT < SET EXPRESSION > |
< ARITHMETIC EXPRESSION > < RELATIONAL OPERATOR >
< ARITHMETIC EXPRESSION > | < SET EXPRESSION >
[ = | EQL | ≠ | NEQ ] < SET EXPRESSION > ;

< RELATIONAL OPERATOR > ::= LSS | LEQ | = | GEQ | GTR | NEQ | # < |
# > | ≤ | ≥ | ≠ | EQL ;

B.6.2 Arithmetic Assignment Statements

< ARITHMETIC ASSIGNMENT STATEMENT > ::= list [ < * I >
[ #[ < SUBSCRIPT LIST > #] ]& ← ]
< ARITHMETIC EXPRESSION > ;

< ARITHMETIC EXPRESSION > ::= < GLOBAL PRIMARY > | < IF CLAUSE >
< ARITH BOOL EXPRESSION > ELSE < ARITH BOOL EXPRESSION > |
< ARITH BOOL EXPRESSION > ;

< GLOBAL PRIMARY > ::= [ FOR (< SET VARIABLE LIST >) SIM
(< SET NAME LIST >) ]& [ SUM | PROD | GOR | GAND |
MAX | MIN ] < ARITHMETIC EXPRESSION > ] ;

< ARITH BOOL EXPRESSION > ::= < ARITH BOOL IMPLICATION >
[ EQU < ARITH BOOL IMPLICATION > ]* ;

< ARITH BOOL IMPLICATION > ::= < ARITH BOOL FACTOR >
[ [ WOR | WEOR | WIMP | WEQV ] < ARITH BOOL FACTOR > ]* ;

< ARITH BOOL FACTOR > ::= < ARITH BOOL SECONDARY >
[ WAND < ARITH BOOL SECONDARY > ]* ;

< ARITH BOOL SECONDARY > ::= < SIMPLE ARITHMETIC EXPRESSION >
[ WNOT < SIMPLE ARITHMETIC EXPRESSION > ]* ;

< SIMPLE ARITHMETIC EXPRESSION > ::= [ + | - ]& < TERM >
[ [ + | - ] < TERM > ]* ;

< TERM > ::= < FACTOR > [ [ × | / | DIV ] < FACTOR > ]* ;

< FACTOR > ::= < PRIMARY > [ #* < PRIMARY > ]* ;

< PRIMARY > ::= < *I > [ #[ < SUBSCRIPT LIST > #] ]& |
(< ARITHMETIC EXPRESSION >) | < MODFUN > |
< FUNCTION DESIGNATOR > (< ARITHMETIC EXPRESSION >) ;

< SUBSCRIPT LIST > ::= list [ < ARITHMETIC EXPRESSION > |
< SET EXPRESSION > | < > ] separator , ;

< MODFUN > ::= MOD (< ARITHMETIC EXPRESSION > ,
< ARITHMETIC EXPRESSION >) ;

< FUNCTION DESIGNATOR > ::= ABS | SIGN | SQRT | TRANS | SIN | COS |
ARCTAN | LN | EXP | ENTIER ;

B.6.3 Set Assignment Statements

< SET ASSIGNMENT STATEMENT > ::= list [< * I > ← ]
< SET EXPRESSION > ;

< SET EXPRESSION > ::= < SIMPLE SET > | < IF CLAUSE > < SIMPLE SET >
ELSE < SIMPLE SET > ;

< SIMPLE SET > ::= < SET PAIR > [ < SET PAIR > ]* |
REVERSE < SET PAIR > ;

< SET PAIR > ::= < SET UNION > [PAIR < SET UNION > ]* ;

< SET UNION > ::= < SET INTERSECTION > [ [UNION | DELETE | SMD |
CONCAT] < SET INTERSECTION > ]* ;

< SET INTERSECTION > ::= < SET FACTOR > [INTERSECT < SET FACTOR > ]* ;

< SET FACTOR > ::= < SET OFFSET > [COMPLEMENT < SET OFFSET > ] * ;

< SET OFFSET > ::= <SET PRIMARY > | (< SET PRIMARY > [+ | -]
< ARITHMETIC EXPRESSION >) ;

< SET PRIMARY > ::= < SET IDENTIFIER > | (< SET EXPRESSION >) |
CSHIFT (< SET EXPRESSION >, < ARITHMETIC EXPRESSION >) |
< SET DEFINITION TAIL > ;

< SET IDENTIFIER > ::= < * I > ;

## APPENDIX C: A METALANGUAGE FOR SPECIFYING SYNTAX

The TRANQUIL syntax in Appendix B is specified in a form of BNF which is extended as follows:[8]

1) Kleene star:

   < A >* = < > | < A > | < A > < A > | ...

   where < > represents the empty symbol

2) Brooker and Morris' question mark (here &):

   < A > & = < > | < A >

3) List Facilities

   list < A > = < A > < A > *

   list < A > separator < B > = < A > [< B > < A >] *

4) Brackets

   < T > ::= [< A > | < B > | < C > ] < D >

   is equivalent to

   < T > ::= < R > < D >

   < R > ::= < A > | < B > | < C >

5) Metacharacters:

   A sharp (#) must precede each of the following characters when they belong to syntactic definitions:

   #, [,], *, ;, <, >.

In the syntax < * I > is used to designate an identifier and < * N > is used to designate a number. Further, the special words in the language are capitalized and underlined.

FEB 7- 1974

BOUND BY
HNM

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R v.3 C002 v.308-315(1969)
TRANQUIL : a languege for an array proce
3 0112 088404311